Turing, His Machine and Computability

Jürg Kohlas
E-mail: juerg.kohlas@unifr.ch

December 17, 2010

Everyone who taps at a keyboard is working on an incarnation of a Turing machine.
Time Magazine, 1999

1 Von Neumann and Turing

In the last number of Philamath John von Neumann is stated to have said, that if somebody precisley describes what a computer can not do, then he would build a machine which does exactly that. Von Neumann surely knew better. In the year 1936 the young British mathematician Alan Turing published the paper "On Computable Numbers with an Application to the Entscheidungsproblem". In this paper Turing defined what it means to compute mechanically and showed that there are real numbers which can not be computed. Based on this result many other precisely defined problems can be stated which can never be solved by mechanical computing. Von Neumann knew Turings paper and even met Turing at the Institute of Advanced Study in Princeton, where he participated in the design of the EDVAC the first computer with the architecture named after him. So von Neumann must have known that not every precisely stated mathematical problem can be solved by computer.

2 Fundamental Questions of Mathematics

At the beginning of the twentieth century Mathematics became more and more formalized. The need arised to base proofs on solid fundaments and to formulate them in a way such that their correctness can be tested any time, at best mechanically. Precise rules for feasible steps of proofs were formulated. Proofs became logical computations which should terminate in a finite number of steps. Even before computer existed, this puts Mathematic already into the scope of computing. Turing, born 1912, attended at the Kings's College in Cambridge a lecture by Newman about Hilbert's program formulated a the Mathematics Congress 1928 in Koenigsberg: (1) Is Mathematics (or some specific part of it) *complete*, can every mathematical statement either be proved or disproved? (2) Is Mathematics (or some specific part of it) *consistent*, are there no contradictory statements which can be proved? (3) Is Mathematics (or some specific part of it) *decidable*, is it possible for every mathematical statement to decide whether it can be proved or not? Hilbert was convinced that every one of the three questions can be answered in the affirmative. It is an irony of history that at the same congress Gödel proved that Arithmetics is either *inconsistent* or else *incomplete*. This answers the first two question of Hilbert in the negative.

The third question (the *Entscheidungproblem*) remained unanswered so far. Turing became fascinated by the idea of mechanical computing. With youthful boldness he attacked the third Hilbert problem.

3 Turing Machine

For Turing a *computer* was a person (in general a girl) computing with paper and pencil. He designed his mechanical scheme of computing after this model: A computation proceeds in steps. At any step the mind of the computer is in one of finitely many possible states, and considers some given symbols on the paper. According to the mind state and the perceived symbols the computer modifies the content of the paper by possibly eliminating old symbols and writing new ones. In addition the state of mind may change too. Accordingly his machine consists of a *control unit* representing the brain of the computer and an infinite tape divided into cells (see Figure 3.1), representing the paper. Into the cells the machine may write symbols from a finite alphabet. Without any loss of generality we may assume a binary alphabet with the symbols "0" and "1". We use further the symbols "Δ" to indicate an empty cell. A read/write head is positioned at any step at exactly one cell and can read and write into this cell. This is called the actual cell. At any step only the following operations are possible: (1) changing the state of the control unit, (2) eliminating the symbol in the

Figure 2.1: Above left: Turing, right: Hibert, below: Gödel in the midth between Poincaré and Kolmogorov.

actual cell, writing a symbol into the actual cell, (3) moving the head one cell to the left or to the right or leaving it at its actual position. At any step each of the three operation is fully determined by the actual state of the machine and the symbol in the actual cell. These are the transition rules which determine the operation of the machine from step to step.

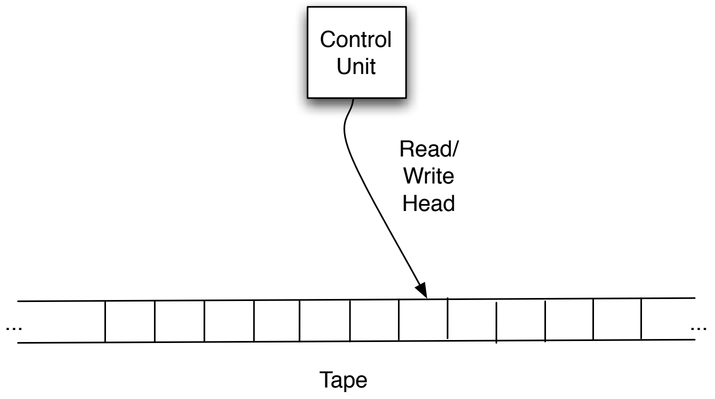

Figure 3.1: Turing's machine: The control unit is at any time in determinate state. The read/write head is represented as an arrow which links the control unit to the tape

The transition rules are best written into a table. We illustrate this by a simple example from Turing's paper: The machine should write the infinite binary sequence 0101010... on the tape into the cells with even numbers, starting at cell number 0. Turing assumed four states q_1 to q_4 for the control unit. For the head move R means a move to the right. Here is the table which defines a machine writing the required binary sequence:

Acutal State	Acutal Symbol	New State	New Symbol	Head Move
q_1	Δ	q_2	0	R
q_2	Δ	q_3	Δ	R
q_3	Δ	q_4	1	R
q_4	Δ	q_1	Δ	R

At the beginning the head is on cell 0 and the control unit in state q_1 and the tape is assumed to be blank. According to first line state changes into q_2, symbol 0 is written into cell 0 and the head moves to the right, to cell 1. Now, the second line applies, which changes state into q_3 and moves the head to cell 2. Now its the turn of line 3 in which symbol 1 is written into the actual cell 2 and state changes to q_4. Next line 4 applies in which state is changed back to the initial state q_1 and the head is moved to cell 4. Now the same sequence of rules (or lines) applies ad infinitum.

It was a particularity of Turing's "programming style" that he used the odd cells as working space and the even ones as output. In the simple problem above the working space is not used. But Turing gives a second example: Generate the sequence 001011011101111011111.... This is rather more complicated to solve and the workspace becomes essential. In modern textbooks, Turing machines are defined slightly different: Among the states of the control unit there are one or more *terminal states* in which the machine stops. This does not necessarily mean that a computation comes into a terminal state and stops. But the possibility exists, whereas Turings original machines in principle never stop. However Turing noted that there may be machines which at some point stop writing into the even (output) cells and write subsequently only into odd cells. These machines he calls *circular*. The *circle-free* machines in contrast output infinite binary sequences. These differences between Turing's original view and the modern view are however by no means essential.

4 The Church-Turing Thesis

Turing very carefully argued in his paper why his machines really compute everything which reasonably can be called *computable*. His view was reinforced when he became aware of a parallel work of the American logician Alonzo Church, who with a very different system, the λ-*calculus*, came to similar conclusions as Turing regarding the Entscheidungsproblem. In fact Turing immediately showed that the λ-calculus and his machines can solve exactly the same class of problems, have the same power. He did this by simulating the λ-calculus on his machine. Later several other systems where proposed which proved to have exactly the same power as Turing machines. Even modern Computers or programming systems have the same power, no more, no less. No computer can solve problems which the Turing machine can not. This became to known as the *Church-Turing Thesis*:

> Turing machines or any equivalent system define what mathematically is meant by an effective or algorithmic procedure or a computation.

It is widely accepted in Computer Science that Turing machines capture what *computing* at base means and that no computer in the near or far future is to be expected to solve problems which Turing machines cannot. This is confirmed with the newest model of computing, with quantum computers. Note however that with Turing machines, so far, we do not talk about time and speed. Nor do we consider ease of programming. That is another issue. In other words, we do not address issues of efficency. Turing machines are not made to practically program complex problems.

5 Gödelization

So, a Turing machines outputs in the even cell a finite or infinite binary sequence. The latter can be considered as the binary expansion of real number. The question arises then, is there for every real number a Turing machine to compute its binary expansion? More precisely, a real number in the interval from 0 to 1 is *computable*, if there is a circle-free Turing machine which outputs its infinite binary expansion on the even cells of the tape. Note also that rational numbers may be represented by infinite exanpansion, for instance 1/2 is either .100000... or .011111.... To simplify here a bit, we limit ourselves to real number in the unit interval. The concept can easily be extended to all real numbers.

In order to answer this question, it is necessary somehow to order or number Turing machines. The technique to do that has already been introduced by Gödel and it is called *Gödel Numbering*. By a usually rather primitive and tedious process each element of a formal system, here each Turing machine, is encoded into an integer. In the box at the end Turing's Gödel numbering of his machines in illustrated with the example above. The number so associated with each Turing machine is called its *description number*. Not any positive integer is a description number, but it can easily be tested whether it is one or not.

This process has far reaching consequences: We may order the description number of all possible Turing machines by increasing value. It is not essental that this is only possible in principle, but not in practice. It means that the Turing machines can be numbered and that the set of Turing machines is *countable*. Since each circle-free machine computes a real number, the set of real number which can be computed is also *countable*. But we know from Cantor that the set of all real numbers is *not* countable! *So there must be* *real numbers which are not computable!* This is clear so far. But can we give concrete examples of uncomputable numbers?

6 Computable and Uncomputable Real Numbers

First, what are then computable real numbers? Surely all *rational numbers* or fractions. Also all irrational *algebraic numbers* like square roots, and so forth, are computable. There are even many *transcendental* irrational numbers like π and e which are certainly computable.

So do we know uncomputable numbers? The tricky question is, what is meant by *knowing a number*? Somehow there must be a mathematical definition of such a number. These definitions must be expressed by sentences of some formal language, for example based on first order logic. Since such sentences are by all means finite, the number of possible sentences is again *countable* and by the same consideration as above, not all real numbers are *definable.* All computable numbers are surely definable, for instance just by their Turing machine. But it turns out that some of the uncomputable numbers are definable too. We are not delving into the necessary formal language, but content ourselves by indication definitions in a more simple language.

The simplest example is the following: Let r_i be the real number computed by the circle-free Turing machine number i, for $i = 1, 2, \ldots$. Construct a real number z by negating the i-th binary digit of r_i, that is, change it to 0 if it is 1 or to 1 if it is 0. This real number is well defined, but it is uncomputable: It is not computed by machine number 1, since the first bit is changed, neither by machine number 2, since the second bit of its number is changed, and so on. So there is no machine to compute it in the list.

A clever reader may now propose to construct a machine along the following line: Take all numbers from 1 on upwards, check each number whether it is a description number and count the machines found so far. If the i-th description number is found, use the machine to compute the i-th bit, change it, and go to the next number... This seems at first sight to be a sound algorithm to compute exactly the number z we claimed to be uncomputable. So where is the hick? The problem is that we must in this procedure for each description number test whether each machine we find is circular or circle-free. It turns out that this is computationally impossible. The problem is essentially equivalent to the famous *halting problem*: Given a (modern)

Turing machine (with terminating states) and some symbols on the tape as input, decide whether the machine stops on the given input. Turing used a diagonal element in the same spirit as Cantor, to show that no Turing machine can decide whether a machine is circular or not. That's why the procedure does not work.

This allows to define a second uncomputable number h: define its i-th bit to be 0 is machine number i is circular and 1 if is is circle-free. Again u is well defined, but uncomputable. In general the indicator function of any subset S of natural numbers which is undecidable, that is, for which there is no Turing machine to decide whether a natural number i belongs to S, defines a uncomputable number. This illustrates also the relation between computable/uncomputable numbers and decidable/undecidable sets, that is the *Entscheidungsproblem*.

Do we now really *know* these numbers by the above descriptions? One might well argue that a number is not really known, if one cannot compute it. There are different levels of knowledge! But, any way, here are some well and precisely defined problems for which not even John von Neumann could have built a computer.

7 A Bit More About Turing

Another major achievement of Turing in his paper on "Computable Numbers" was the insight that there is a *universal Turing machine*. Each individual Turing machine solves a particular problem, computes possibly a particular real number. But a Turing machine U can be built which simulates every other Turing machine in the following sense: If the description of a Turing machine T , for instance its description number, is written on the tape, then machine U computes the same output as machine T. This is like a universal computer which can execute any program and solve thus an (computable) problems with one machine. In fact there are many universal Turing machines. This insight of Turing was an important motivation for von Neumann in the design of the EDVAC.

During the war years Turing got engaged in *Bletchley Park* in cryptology, where he particitpated in decoding the German Enigma and Fish codes. This activity remained secret until the seventies. Turing's contributions in this field are regarded as a decisive part in the U-boat war. After the war Turing engaged into the construction of computers and studied morphogenese. His homosexuality caused in those times a lot of problems, in particular in the context of his part in secret services. This was the tragic of his life. He died in 1954, presumably by suicide, through a cyanide poisoning. Time Magazine elected Alan Turing among the hundred most important personalities of the twentieth century, and this in acknowledgement of his fundamental contribution to Computer Science.

I refer the reader, who wants to know more, to the excellent biography by Andrew Hodges *Alan Turing, the enigma*, Vintage 1983. You may also consult also the web site of Manfred Börgens http://homepages.fh-friedberg.de/boergens/philatelie.htm for additional information on the subject.

Turing's Gödel Numbering of his Machines.
A Turing machine is defined by the table of its transition rules. And the lines of this table have the form $q_i s_i q_j s_j M$ designing acutal state and actual symbol, new state and new symbol and head move, where $M = R, L, N$ (R for Right, L for Left and N for No move). The whole table can be written as sequence of those quintuplets, spearated by a semicolon: $q_i s_i q_j s_j M; q_k, \ldots$. Next, in this string, each q_i is replaced by the letter D followed by i times of letter A and each s_j by D followed by j letter C. The blank is assumed to be represented by s_0, the symbol 0 by s_1 and 1 by s_2. Now the description of the Turing machine is coded by a sequence of letters D, A, C, R, L, N and the semicolon "$;$". Finally Turing replaces A by the number 1, C by 2, D by 3, R by 4, L by 5, N by 6 and "$;$" by 7. The result is an integer, called the *description number* of the machine.
Consider as an example the machine defined by the transition table above. It is represented by the string

$$q_1 s_0 q_2 s_1 R; q_2 s_0 q_3 s_0 R; q_3 s_0 q_4 s_2 R; q_4 s_0 q_1 s_0 R;$$

This string is translated into the letter code

$$DADDAADCR; DAADDAAADR;$$
$$DAAADDAAAADCCR; DAAAADDADR;$$

from which we obtain finally the description number

$$31331132473113311134731113311113224731111331347$$

Note that this procedure is reversible: From the description number we may reconstruct the table of transition rules of the machine by inversing the coding process!